High Performance Computing - HW 6

Anthony Maylath

May 11, 2019

The Courant Institute for Mathematical Sciences, New York University

1 Question 1

My implementation can be found *jacobi_mpi.cpp*. The implementation takes two command line arguments: the first sets the dimension of the problem, N, and the second specifies the max number of Jacobi iterations. My implementation uses row major ordering for the solution lu. I send a different communication depending on where the process falls on the grid. If the process computes the solution on the top row, then don't comunicate ghost values for the top. I filter out top row comunication by checking if $mpirank >= \sqrt{p}$ where p is the number of processes. I apply similar checks on all four edges of ghost values.

Figures 1 and 2 demonstrate the weak scalling of the implementation for 64 nodes with 4 cores per node and 16 nodes with 16 cores per node respectively. Each line represents a fixed number of Jacobi iterations. The base size of the problem is $N = 100$ and is scalled up proportional to the number of processes so that each process gets $lN = 100$ locally. The secondary axis corresponds to 1,000 and 100 iterations. For 100 iterations, it seems like we lose some performance for larger problem sizes, while the performance time seems more consistant across problem sizes for more iterations.

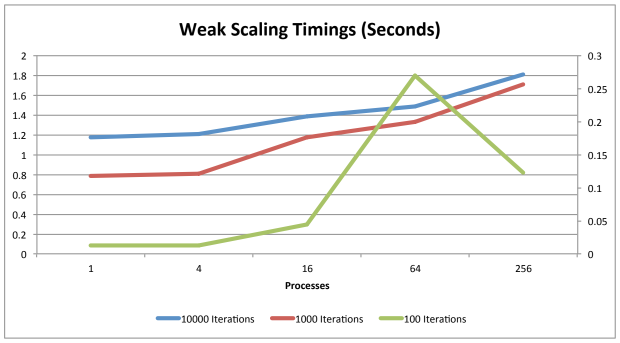

Figure 1: Weak Scaling 64 Nodes, 4 Cores Per Node

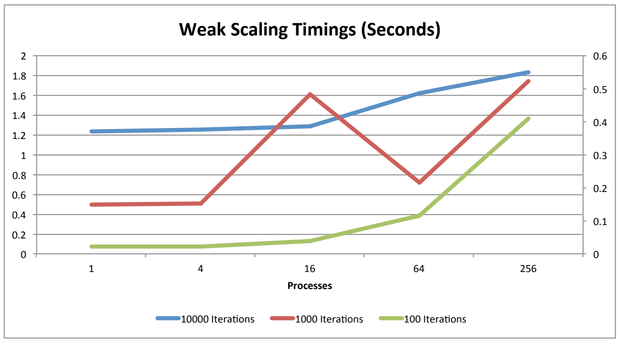

Figure 2: Weak Scaling 16 Nodes, 16 Cores Per Node

2 Question 2

sample_sort.cpp contains my implementation of sample sort with mpi processes. I start by assigning each process a list of integers of size $lN = N/p$ where N is specified as the first command line argument. The local lists are initialized with the $rand()$ function in the standard library. The spliter candidates are selected to be the first $p-1$ elements of each local list, where p is the total number of processes. Since the numbers are randomly initialized, these first elements will have random order. To perform local sorting, I use quicksort. The implementation is located in header file, $hw6helper$, and is a modification of a quicksort algorithm from stack overflow. The sorted output is stored in files named $output(rank).txt$ where rank is the local mpi rank. Each file has elements in ascending order and files with larger ranks contain larger elements.

I ran my timings on Prince with 16 nodes, 16 tasks per node and 16GB of memory. Figures 3 and 4 graph the strong and weak scaling respectively. Each symbol in 3 represents a list size. We see larger list sizes give better strong scaling. Also, the performance seems to go down if we use too many processes. The optimal processes seems to be 16 and 50 for list size 1 million and 10 million respectively. Perhaps we need a larger list size to see optimal performance with over 100 processes. Figure 4 keeps the local list size as 10,000. The global size is increased with the number of processes. The run time goes up as we increase the problem size, except when we move from 4 to 10 processes when it goes down. The trend line indicates a slope of 0.0148 which implies that about 0.02 seconds are added to the run time when the list size is increased by 10,000 and the number of processes are increased by 1. The R^2 of the trendline is quite high at 0.998 which suggests the weak scaling really is linear; however, we probabaly need more datapoints to make the result statistically significant.

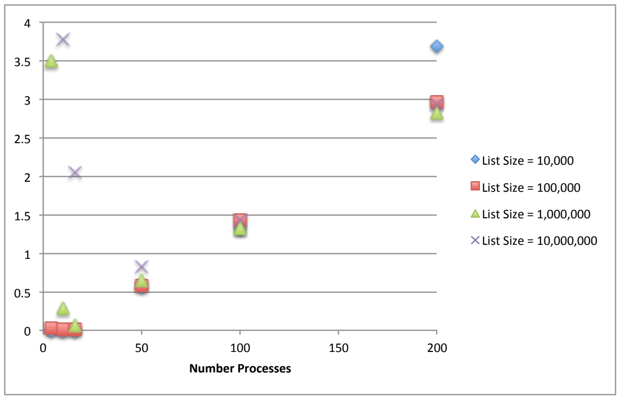

Figure 3: Sample Sort: Strong Scaling

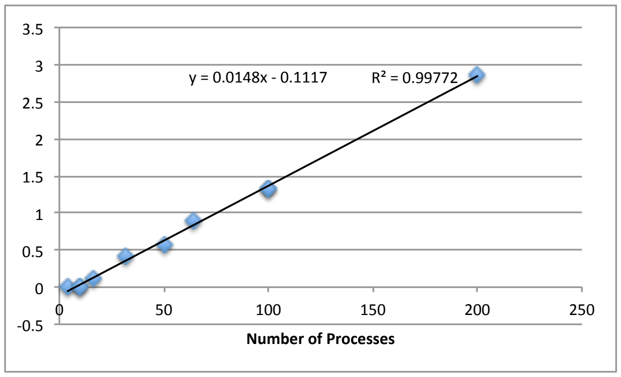

Figure 4: Weak Scaling for 10,000 Points per Process